ARI Research Note 96-07

General Principles for an Intelligent Tutoring Architecture

John. R. Anderson, Albert Corbett, Jon Fincham,
Donn Hoffman, and Ray Pelletier
Carnegie Mellon University

Research and Advanced Concepts Office
Michael Drillings, Acting Chief

November 1995

19960311 128

United States Army
Research Institute for the Behavioral and Social Sciences

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 5

U.S. ARMY RESEARCH INSTITUTE FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency Under the Jurisdiction of the Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON
Director

Research accomplished under contract
for the Department of the Army

Carnegie Mellon University

Technical review by

Joseph Psotka

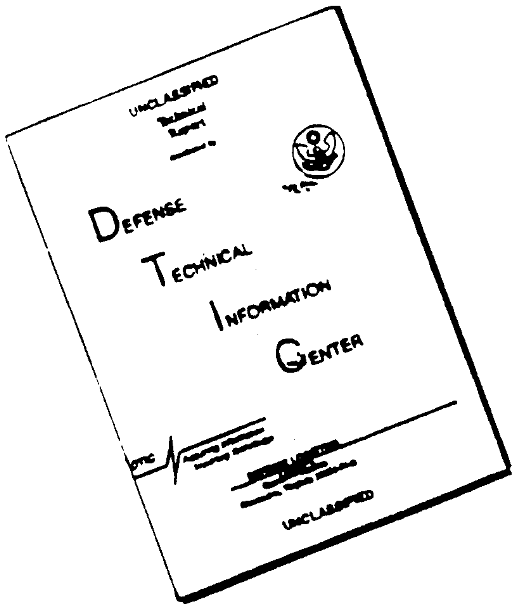
NOTICES

DISTRIBUTION: This report has been cleared for release to the Defense Technical Information Center (DTIC) to comply with regulatory requirements. It has been given no primary distribution other than to DTIC and will be available only through DTIC or the National Technical Information Service (NTIS).

FINAL DISPOSITION: This report may be destroyed when it is no longer needed. Please do not return it to the U.S. Army Research Institute for the Behavioral and Social Sciences.

NOTE: The views, opinions, and findings in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other authorized documents.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

REPORT DOCUMENTATION PAGE

1. REPORT DATE 1995, November	2. REPORT TYPE Final	3. DATES COVERED (from. . . to) July 1985 - July 1989
4. TITLE AND SUBTITLE General Principles for an Intelligent Tutoring Architecture		5a. CONTRACT OR GRANT NUMBER MDA903-85-K-0343
		5b. PROGRAM ELEMENT NUMBER 0601102A
6. AUTHOR(S) John R. Anderson, Albert Corbett, Jon Fincham, Donn Hoffman, & Ray Pelletier (Carnegie Mellon University)		5c. PROJECT NUMBER B74F
		5d. TASK NUMBER 711C
		5e. WORK UNIT NUMBER C02
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University Pittsburgh, PA 15213		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Institute for the Behavioral and Social Sciences ATTN: PERI-BR 5001 Eisenhower Avenue Alexandria, VA 22333-5600		10. MONITOR ACRONYM ARI
		11. MONITOR REPORT NUMBER Research Note 96-07

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

COR: Judith Orasanu

14. ABSTRACT *(Maximum 200 words)*:

This report describes the major outcome of our research project which has been a set of ideas for developing intelligent tutoring systems and an architecture for implementing these ideas. The approach is built around developing a production system model of the skill being taught. Declarative instruction is built to communicate the production rules, a model tracing methodology is implemented to monitor their learning, and a knowledge tracing methodology is implemented to guarantee their mastery. The multiple programming languages project was an attempt to build a single architecture based on these ideas which was capable of teaching many different programming languages. It has been used so far to teach LISP, prolog, and pascal at CMU, and NYNEX has adapted it to teach COBOL. Current research is aimed at building tools to extend this architecture to an authoring system for intelligent tutors generally.

15. SUBJECT TERMS

Intelligent tutoring architecture

SECURITY CLASSIFICATION OF			19. LIMITATION OF ABSTRACT	20. NUMBER OF PAGES	21. RESPONSIBLE PERSON (Name and Telephone Number)
16. REPORT Unclassified	17. ABSTRACT Unclassified	18. THIS PAGE Unclassified	Unclassified	29	

Table of Contents

Abstract

This report describes the major outcome of our research project which has been a set of ideas for developing intelligent tutoring systems and an architecture for implementing these ideas. The approach is built around developing a production system model of the skill being taught. Declarative instruction is built to communicate the production rules, a model tracing methodology is implemented to monitor their learning, and a knowledge tracing methodology is implemented to guarantee their mastery. The multiple programming languages project was an attempt to build a single architecture based on these ideas which was capable of teaching many different programming languages. It has been used so far to teach LISP, prolog, and pascal at CMU and NYNEX has adapted it to teach COBOL. Current research is aimed at building tools to extend this architecture to an authoring system for intelligent tutors generally.

We have developed a number of intelligent computer-based tutors for the domains of LISP programming, geometry theorem-proving, and algebra. The state of this research as it stood in 1987 is summarized in Anderson, Boyle, Corbett, and Lewis (1990). These tutors had modest success in real classroom situations producing improvements on the order of one standard deviation or one letter grade. More recently we have been trying to identify the essential features of our tutoring methodology, the theoretical bases for these features, and have been trying to develop a tutoring architecture that facilitates creation of tutors with these features. This report identifies what we feel are the core principles for intelligent tutoring; describes a first-pass architecture that partially embodies these principles, describes our experiences in implementing tutors for LISP, Prolog, and Pascal in this architecture, and describes the current status of our work on creating a new system which is an authoring system for tutors of this general type.

We call our approach to tutoring a model-tracing methodology. This involves first developing a cognitive model which is capable of solving problems in the same way we want students to solve the problems. This cognitive model is then used to interpret the student's performance of some task at the computer. Basically, the tutor tries to find some way of solving the problem within the cognitive model that matches the student's problem solving and uses this way as the interpretation of the student. All instructional decisions are driven off this interpretation. In our view the success of our tutoring methodology is basically a result of our use of a cognitive model. This success derived from basic principles of skill acquisition as embodied in theories like the ACT* theory (Anderson, 1983, 1987). In the next section of the paper, we describe the basic implications of that theory for tutor design. These implications turn out to be surprisingly simple.

Key Elements of the Model-Tracing Methodology

According to the ACT* theory a cognitive skill is represented as a set of production rules. There is considerable evidence for this assumption (e.g., Anderson, 1983, 1987; Just & Carpenter, 1987; Kieras, 1982; Kieras & Bovair, 1986; Newell & Simon, 1972; Singley & Anderson, 1989) and the general success of our approach to tutoring can be seen as further evidence. A production rule is a condition-action pair which specifies taking some problem-solving action when a certain condition is met. We have found that the skills being taught in a course can be decomposed into many hundred production rules.

Working within a production system architecture like ACT* places a lot of constraints on the representation of a skill. The necessity of representing the skill as productions provides some constraint but much further constraint comes from style rules (Bovair, Kieras, and Polson, 1990) imposed by the particular theory. However, for complex tasks it is not the case that there is an unique way of representing the skill. Thus, for instance, consider the skill of writing extractor sequences in LISP. An example of an extractor sequence is (car (cdr (cdr lis))). LISP evaluates such embedded sequences inside out and so will twice apply cdr to take the tail of the list and then car to extract the next element. So if lis were (a b c d) it would return c. Notes that the order in which the LISP functions (car and cdr) are listed is just the opposite of the order in which they are evaluated. One can write productions which plan this code in the left to right order that they are written or in the right-to-left order that they are evaluated. In the former case, the first production to apple would be:

```
P1        IF    the goal is to get the nth element of the list
        THEN    use car as the first function
                   and set as a subgoal to code an argument to car
                   that will get the n-1st tail of the list
```

In the latter case the first production to apply would be:

```
P2        IF    the goal is to get the nth element of the list
        THEN    use cdr as the most embedded function
                   and set as a subgoal to get the n-1st element
                   of that list
```

The first production rule would guide coding as in our original LISP tutor while the second production rule would guide tutoring as in the GIL tutor of Reiser, Ranney, Lovett, and Kimberg (1989). The theory provides no direct guidance on the issue of which method to teach to students or neither.

However, there can be important consequences of the model taught. Students may find one method more in keeping with their prior methods. Thus, in the above case one could either argue that the first method is consistent with left-to-right problem-solving or the latter method is more in keeping with forward causal reasoning. It remains an empirical issue to evaluate the two. Also, one method might just be more powerful than another. Thus, for instance, Koedinger and Anderson (1990) present a model of geometry problem solving which is capable of solving more difficult problems than the cognitive model embedded in the original geometry tutor of Anderson, Boyle, and Yost (1985) Another possibility is that one method leads to greater transfer. Thus, Singley, Anderson, and Gevins (1989) describe a model for solving algebra word problems which leads to greater transfer than the method typically taught in algebra textbooks. Perhaps the most important issue about choice of cognitive model is that there can be a clash between the method advocated by the tutor and the methods taught elsewhere to the students. Thus, a major source of difficulty in our work with the algebra tutor was that the methods it employed, based on one textbook (Keedy, Bittinger, Smith, 1978), conflicted with the methods students had learned in the classroom.

In our view development of the cognitive model is the most important aspect of tutor development and the most time intensive. It is basically the problem of developing an expert system to solve a problem with the added constraints that the system solve the problem in a cognitively plausible way, that it satisfy some measure of optimality among alternative methods, and that it be consistent with the methods being taught to the student elsewhere. It is not a task that can be done well without intensive study of the domain and the context of its instruction. It is a task which no amount of prior cognitive theory nor development of authoring tools will eliminate.

The work and difficulty involved in developing an adequate cognitive model is the bad news in this approach to tutoring (see Bovair, Kieras, and Polson, for similar comments). The good news is that once this is accomplished the rest of the task is relatively easy. There is a lot of support that can be offered for it and a very high probability of significant improvement in student achievement.

Declarative Instruction

Developing a production rule model amounts to identifying the instructional objectives. Each production rule is another piece of knowledge that one wants to communicate to the student and have the student master. The ACT* theory provides strong guidance on the issue of how these production rules are to be communicated. According to the theory production rules are acquired by analogy to examples of solutions that were produced by these productions. Thus, if we wanted to teach a student productions P1 or P2 above we should center our instruction around an example involving this production. Thus, we might show them the example

```
(car (cdr '(a b c)) = b
```

The critical issue is not just presenting the example but to attach to it information to explain how the example relates to problem solving goals. Essentially we need to attach to the example the production rule it is supposed to illustrate. So if we were using the example above to teach P1 we would need to also present P1 and see if the student understood the application of P1 to the example. We advocate doing this through a series of questions that make sure the subject can map each clause of production onto the example:

What is the goal of this code?
answer: to get the second element of (a b c)

What function do you use?

answer: car

What is the function's argument?
answer: (cdr '(a b c))

What is the goal of the argument?
answer: to get the first tail of (a b c)

In our view appropriate declarative instruction consists of presenting the production rules in English, providing examples of the application of these rules, and interrogating the student so that the student understands the application of these rules to the example.

Model-tracing Practice

Once such rules have been explained the next step is to have the student solve problems which involve the target production rules. There are two goals in such problem-solving. One is to determine if the declarative instruction has really been properly encoded. The second is to give the student opportunity to compile their declarative knowledge into production rule format and practice that. Both goals require that we be able to interpret the student's problem-solving behavior and identify what rules they are applying, correctly or incorrectly. This is where the model-tracing methodology becomes involved. Simply put, we try to find some sequence of production rules in the underlying student model which will reproduce their behavior. If we can, then we give them credit for understanding. If not we need to find some best interpretation of their behavior which will allow us to determine the points of discrepancy. These points of discrepancy will be points where a particular production rule should have fired but did not. That failed production rule becomes a target for further instruction and remediation. Besides needing to deal with errors in problem solution the tutor needs to be able to help the student when they are stuck. This again requires interpreting the students current problem solution and determining what production should apply next. That production becomes a target for advice.

The task of model tracing is difficult because of ambiguity--more than one sequence of production rules could have produced a particular surface behavior. This makes it impossible to proceed with any certainty and creates computational problems as we need to follow a potentially exploding number of alternative interpretations. Our methodology develops its distinct character because of the strong measures we take to tame that ambiguity. One measure is to insist that a student never deviate from a correct solution path as we do in our immediate feedback tutors such as described in Anderson, Boyle, Corbett, and Lewis (1990). Another measure is to present disambiguation menus to students as soon as they produce behavior that can be generated by more than one correct production rules. A third measure is to try to impose strong stylistic constraints on the student to restrict the number of acceptable solutions. Such measures have unfortunate negative side effects and one of the goals in the research that we will describe has been to find ways to maintain the interpretability of behavior but avoid these negative side effects.

The system needs to be able to respond to holes in the problem solution, whether these holes reflect overt errors or points where the student simply cannot progress. Our method is to essentially reinstruct the needed production rule. There is a danger in simply representing the rule as that may provide the student with more information than they really need. There is ample evidence that people remember better what they can generate for themselves rather than what they are told (Anderson, 1990b). It is also possible that processing elaborate feedback will interfere with problem solving. A final danger is that subjects will process the feedback we present them just to extract the answer and not really understand why it is the answer. Thus, we have adopted a successive questioning strategy in which we provide students with minimal information and then only more as needed.

One could imagine a scheme in which there were many layers of progressive hints, but our students find such a scheme frustrating. Rather we have opted for a two-hint scheme in which the first level basically frames the task and leaves it to the student to solve it while the second level explains the correct solution. Thus, for P1 we would present at the first level:

"You need to come up with some sequence of extractors that will produce the third element of a list. Remember that LISP evaluates its functions inside out."

While the second level we would present:

"To get the third element of a list you need to get the second tail of the list and then apply car to get the first element of that list. Since LISP evaluates its functions inside out, the first thing you will code is car and then you will code its argument which gets the second tail."

An interesting aspect of this approach to tutoring is that it places no value on bug diagnosis and remediation which has been the traditional heartland of intelligent tutoring research (e.g., Sleeman & Brown, 1982). The reason is obvious--the student's problem is that they don't know the correct rule and this is what needs to be repaired. They really do not need an elaborate explanation about what peculiar mental state led to error. There is now research finding that reinstruction helps while bug remediation does not (Sleeman et. al., 1989).

There needs to be one strong qualification placed on this rejection of bug remediation. A bug needs to be remediated if it actively interferes with student's incorporating the correct instruction. There are two ways this can happen. First, the subject can have some misconception that causes the student to systematically misinterpret the instruction. This appears not to be a great difficulty in the domains of programming and mathematics where students do not harbor elaborate and strongly held misconceptions about the subject matter. It may will be a serious issue in other domains like physics where strong misconceptions have been shown to interfere with learning the target domain (McCloskey, 1983). The second way misconceptions can interfere is that the student simply refuses to process the instruction convinced that he or she is right. In contrast to the first category of distorting misconception, this category of obstinate misconception is quite prevalent in the domains of programming and mathematics. Here the students really need not have their misconceptions explained, they only need be convinced that their belief is incorrect. Sometimes, a simple "error diagnostic" rather than a bug message is sufficient. Thus, if the student enters "write" rather than "writeln" for PASCAL, we will deliver the message "You need to issue a line fed and write does not do this." On other occasions a certain exploratory component to the tutor is useful. Students can try out the code they believe in and see that it does not produce what they believe it will or students can try to create a model of their mathematical solution and see the contradiction.

Knowledge Tracing and Mastery Learning

The outcome of the model tracing process is a scoring of the production rules in a problem solving episode as to whether they were performed correctly or not. We can use this scoring to estimate the probability that the student has mastered the target production rule. Over problems we can keep updating our estimates of what the student knows. This is what we call knowledge tracing in contrast to model tracing. We have developed an algorithm for performing such knowledge tracing based on work of Atkinson (1972). Our application of their technique is described in AI (Corbett, Anderson, & Patterson, in press).

There are a number of things one can do with this interpretation of the student's knowledge state. One is to simply communicate this interpretation to the student or teacher which we believe can be very useful. The more elaborate use we make of it within our tutors is to implement a remediation algorithm in which we select problems designed to practice students on those productions they are weak on. We divide the material up into a large number of curriculum units where each unit involves introducing the student to a small number (less than 10) of new related production rules. As in standard mastery-based methodology (Block, 1971), we try to assure the students have mastery of the elements in the current curriculum unit before promoting them to the next unit. We continue to present to the students remedial problems until all the target production rules exceed a certain threshold defined as a probability of mastery. In our applications we have had pretty good success (Anderson, Conrad, & Corbett, 1990) in using a mastery level defined by a 95% confidence on our part that the student has mastered the target production rule. Given the way we typically parameterize the knowledge tracer, this means that student will have at least 2 to 4 opportunities to perform the target rule. That is to say a couple of successful uses or an error and three successful uses is enough to raise the production rule above threshold.

Summary: The critical intervening learning variable

We have now reviewed the essential ideas involved in creating a model-tracing tutor. They are clearly centered on the production rules and providing students with example problems that illustrate their use. In our theory, the critical intervening variable that determines rate of learning is the number of problems the student has both solved and understood. Each problem solved and understood gives the student another opportunity to compile and strengthen production rules. Note, it does not matter on this view how students actually achieve the problem solution only that a solution is obtained. We have found, in fact, that students following very different trajectories to final solution and understanding are nonetheless equivalent in their resulting problem solving skills (Anderson, Conrad, & Corbett, 1990). This argues that within the constraint of achieving understanding, one wants to do everything possible to maximize the rate of progress through relevant examples. There are really three critical criteria by which one wants to judge instruction. First, one wants the student to achieve understanding. Thus, it would not do to simply tell the student what to do in problem solving and blindly follow the instruction. Second, the problems have to be relevant to the instruction goals. Thus, for instance, it does no good to give students massive practice on components that reflect only a fraction of the target skill. The third constraint is speed. A pure discovery environment (assuming understanding can be achieved) is a poor idea because some things are very hard to discover without guidance. There is a growing body of data which is consistent with the benefit on instruction that emphasizes these three conditions (Anderson, Conrad, & Corbett, 1989; Black, Bechtold, Mitrani, & Carroll, 1989; Carroll, in press; Carroll & Mack, 1985).

We feel that knowledge tracing achieves the relevancy criterion and that our example-based instruction and incremental-feedback achieve the understanding criterion. The criterion that is actually the most challenging is that of maximizing rate of progress through the problems. Here the challenge is to create an interface that minimizes irrelevant time and maximizes profitable learning. Most paper and pencil or standard computer-editing environments leave too much unnecessary detail for the student to manage. We advocate what we call the structured interface. A prototypical example of a structured interface is the structured editor in programming which is a system that can take care of all the low-level details of syntax in programming without really understanding the goals of the student. More generally, we view the structured interface approach as doing for the student what the technology can provide without the intelligence to solve the problem. Thus, in our word problem tutor (Singley, Anderson, & Gevins, 1990) we provide students with automatic symbol manipulation facilities. In the context of geometry theorem-proving this means providing student with a proof-checking facility among others.

Structured interfaces do for the student some of what is the target of traditional instruction. Thus, structured editors in programming eliminates the need for students to learn syntax. This often provokes the criticism that tutors which use them do not teach the skills that these systems provide--syntax of the programming language, algebraic symbol manipulation, or proof checking as the case may be. We have two responses to such criticisms. The first is to note that these are typically not the skills students have difficulty mastering. The second is that there is no reason for students to master these skills since they can be automatically provided. Thus, we feel that we are justified in providing these computer-automated components if they can accelerate the acquisition of those problem-solving components that cannot be automated.

The Multiple Programming Languages Project

Next we would like to discuss the initial results from our multiple programming languages project which is an approximate attempt to embody the philosophy outlined above and which serves as the basis for our projections for the new authoring system. This system used a common production system, interface, and tutoring architecture to teach LISP, Prolog, and Pascal. We will illustrate it with respect to Pascal.

Figure 1 shows how the system appears when a Pascal problem is first brought up. At the top there is a scrollable window with problem statement, below it another scrollable window with a code template, and below it a window for messages from the tutor, and below it a window for typing in information. Off to the right is a menu of actions that can be taken. A particular node in the code template is highlighted. This is the node that the student will expand by his or her next action. The student can choose to shift the focus to a different node by clicking on the desired node. The student can indicate how to expand the node

Insert Figure 1 About Here

Figure 2 shows the code window at a relatively late point in the process. The student has chosen to expand the highlighted node as a arithmetic expression; a submenu of arithmetic expansions has been brought up, and the student has chosen to expand this as multiply. Most of the code entry takes place by means of menu selection. The student is required to type in names of identifiers and other such terms. There is also a facility by which the student can type in larger pieces of code and a parser will analyze this into a set of menu actions an produce the resulting code. Our model for this interface was taken from the system used here at CMU (Goldenson, 1988; Miller & Chandok, 1989) for the instruction of introductory programming to the bulk of CMU undergraduates. This course uses a structured editor but not a tutor. Considerable success has been reported for it.

Insert Figure 2 About Here

The error feedback in the tutor is quite primitive. We recognize a few common bugs such as when students confuse *write* and *writeln* in Pascal and provide explanatory feedback but most of the time when the student makes an error we simply respond by stating that it is an error. Most of our effort at instruction takes place upon requests for help from the student in which case we present a series of successively strong hints culminating in telling the student what to do. However, the design of the hints was purely a matter of intuition. In redoing the effort we intend to use the two-step hinting procedure illustrated earlier.

The current tutor is an immediate feedback tutor that insists that the student stay on the correct path. However, we view this simply as a holding pattern until we can implement something closer to the version of the LISP tutor which we call the flag tutor (Corbett & Anderson, 1990) In the flag tutor system when a student makes an error, the tutor places the erroneous code in bold to flag that it is an error but the student is allowed to continue coding. This is something that is easy to do in a syntax-based editor. We have observed that 80% of the time when the student's error is flagged the student will spontaneously correct it; 10% of the time they ask for help; and 10 percent of the time they continue coding despite the warning they are in error. Occasionally, students are actually on a correct path of solution and the tutor will accept this if it works. More often students will become convinced they are in error and go back and seek help from the tutor at the point of error.

This flag tutor has the much of the advantage of the immediate-feedback tutor in allowing students to quickly correct mistakes. It minimizes time spent processing feedback and allows students to self correct without any feedback. It also allows students, when they are set on their erroneous solution, to become convinced that they have a problem before processing the tutor's feedback. It also allows the student who knows something innovative to express it. We have found students get through the curriculum in the flag tutor just as fast as with the immediate feedback tutor. They tend to get higher final achievement scores and give more favorable ratings of the tutor, although neither of these results has proven statistically significant.

One of the major technical achievements of the tutor was the use of the Tertl production system. This system was designed to take advantage of the constraints of tutoring in a model-tracing paradigm. In such a paradigm, as long as the student stays on course of an interpretable solution, only one production will fire at each point and that production will apply to a part of the solution the student designates. It turns out in this case a production system can be written which identifies the correct production in the time proportional to the number of productions and independent of the size of problem representation. This means that we are relieved of the complexity barrier and have been able to tutor many hundred line programs. Problem complexity had always been a major limitation in our previous efforts at tutoring.

Another nice technical feature of our system was the ability to enter problems into the tutor as direct code. This facility depended on the parser we developed for processing code. Basically, we attached to the parser the ability to reverse the production rules involved in coding and produce a working memory representation of the problem. This representation had to be augmented with various English referring expressions and certain information about which orderings in the code were critical but it made entering problems much less laborious than previously. It is also an important step towards giving a teacher a facility for problem entry that does not require knowing the intricacies of production rule coding.

Another interesting feature of this multiple programming languages tutor is that it served as the basis for transfer to another cite of another application., The researchers at NYNEX (Gray & Atwood, in press) were able to take this system and adapt it to provide a tutor for COBOL which is taught internally within NYNEX. Their tutor is also used at Metropolitan Life and they are exploring the options of using it at other cites.

Classroom Experience

In the fall of 1989 we completed a classroom test of the tutor in which it taught a semester course to undergraduates in the School of Humanities and Social Sciences at Carnegie Mellon. These students were selected to never have a programming course before. In that semester course, students mastered LISP, Prolog, and Pascal to the point where they could write in each language a program that would solve an arbitrary 8-puzzle problem (Nilsson, 1971). There is no comparable non-tutor course but we view this as a dramatic level of achievement for beginning students. We believe that one bases for the success was the transfer of skills across programming languages. However, the order of learning the programming languages was not systematically manipulated. (It was always Pascal, then LISP, then Prolog). It remains a goal to manipulate language order and see how much the learning of later languages is facilitated.

In the class we experimented with a certain facility which proved to be popular with students and which illustrates a general direction we need to go in developing a general tutoring system. We graphed for students how well they were doing on the various production rules that we were monitoring. Figure 3 shows an illustration for a lesson on recursion. The length of each bar represents our estimate of the probability that the student has mastered each of the rules. Upon each action that the student takes an appropriate production rule will grow or shrink. The line to the right represents the 95 percent confidence threshold that we use for assuming the student has mastered the production rule. Students appreciate this access to the system's internal model of them. In the future we intend to attach declarative, example-based text to each of these bars so that students can bring up instruction relevant to a rule they are having difficulty on. This effort reflects the direction of making the inner workings of the tutor available to student and teacher. We feel that it is important that its behavior be as transparent to all as possible so that the system not appear as a mysterious burden being forced on everyone. Another step in this direction might be to let students chose which rules they wanted remediation on rather than leaving this step in the hands of the tutor.

Insert Figure 3 About Here

The system was not without its difficulties. Besides the glitches associated with new software, students found the option of typing in code and having it parsed tempting but frustrating.[2] It often seemed attractive but they were constantly making errors in their syntax and so would go back to menu based entry.

Students also complained about getting lost in the large programs where the tutor would be providing feedback about the next line of code in some submodule while students had lost track of what the submodule was supposed to do. This suggests the need to provide appropriate explanation to students of what various subgoals where supposed to achieve.

An interesting outcome concerned the use of recognition-based code entry through the menu system. We did a comparison of this with recall based entry as in the LISP tutor where subjects have to recall and type the code in. Students trained on a recognition based system did not do as well as recall subjects when they were asked to do paper and pencil tests. We have subsequently obtained evidence that this reflects a mismatch between the mode of training (recognition) and the mode of the paper and pencil test (recall). If tested in a recognition mode they are actually superior to recall subjects. This raises serious questions about the criterion for which one is training. As we stated earlier we have bought into the view that we are training students to do well in a recognition-based structured editor and so are not necessarily bothered by these results. However, others might take the view that paper-and-pencil recall-based performance should serve as the criterion and would want a recall-based tutor.

The tutor provoked a lot of complaints from those who had to develop software on it. Developing a tutor for a language required the mastery of a lot of baroque details and there were relatively few facilities for debugging the software. As long as such development is done in-laboratory these were tolerable inconveniences. However, it is clear that the development environment will have to be substantially cleaned up if we are to see it used out of the lab.

[2]Actually, we created this system in response to student requests for such a facility. This is one example (among a number) of a feature which students demanded only to complain bitterly when it was provided.

An Authoring System for Intelligent Tutors

We think we now sufficiently understand the model tracing methodology that we can create an authoring system for this style of tutor. There are two related motivations for going in this direction. One is that we would like to extend the range of people who can create such tutors and extend their use beyond laboratory classrooms. The second is that we want to formalize our theory of of tutoring and lay the empirical groundwork for extensive empirical testing of this theory. Our claims of success will always be received with a certain justified skepticism as long as it is we who are testing our own handcrafted tutors. We see this as a necessary step for the field of intelligent tutoring in general. The time has passed when ones ideas should should be given the protection of ones own laboratory and the only basis for deciding among alternative proposals is rhetoric. It is time these ideas get out into the world in a form that they can be explored and tested. It is also critical when this happens that they be subject to rigorous experimental test and not more rhetoric.

In service of this we are trying to create an appropriately portable system, implemented in CommonLISP sufficiently efficiently that it can run on commonly available systems like the MAC II. We need to eliminate the difficulties in use of the system. We also need to release the tight connection our systems have had to a prescribed curriculum and instructional mode. Clearly, in application educators want the freedom to choose what will be taught and how it will be taught. Similarly, researchers want a tool that will allow real variability.

When we talk about an authoring system for intelligent tutors, it is important to appreciate that there are levels of authoring and different users would like to have access to different levels. Our discussion with teachers suggests they most would like control over the problems that are used and the sequence of the curriculum. For them a system that allowed them to enter problems to be tutored and specify the sequence of problems would be adequate. Many teachers are also very concerned about the language the tutor uses and that it be congruent with the language they use in the classroom. As teachers do not agree as to what language is appropriate it is important they can also have access to this without having to master the more technical aspects of the tutor.

As we have discussed earlier, if someone is actually going to construct new educational software in the tutor there is no way to avoid the task of developing production system student models. We think this can be facilitated in various ways and one of our research goals is to study the acquisition of production system modelling skills and perhaps tutor them as we have other skills.

A person doing production system modelling is still quite far removed from the internals of the system. We suspect that the task which will take the software developer closest to the internals in developing the structured interface. In certain applications it may be possible to borrow a structured interface as NYNEX was able to borrow our structured editor interface. However, if one is striking out in a completely new domain one will have to go down to this level which will require mastery of Commonlisp and its relationship to the tutor.

Below we review the steps involved in creating a tutor for a new domain. They are listed roughly in the order that they will have to be addressed in development. One might imagine software developers doing the early steps and end-users such as teachers doing the later steps. With each step we will discuss some of the issues involved and some of the special considerations that arise in these steps with respect to our programming applications. However, for contrastive purposes we will also discuss the application of these steps in the anticipated conversion of the word problem tutor (Singley, Anderson, & Gevins, 1990).

Step 1: Develop an Structured Interface

In our view the first step in developing an interface has to be specifying the structured interface for problem solving. This is a system that can be used without the tutor. One needs to construct and test out such a problem-solving environment before actually developing the tutor. As we see problem-solving it is not independent of the interface in which it takes place and so one cannot develop a cognitive model until one has settled on the interface. Then one can observe problem-solvers with that interface and try to develop a cognitive model. This observation process may also lead to suggestions for improvement in the interface. The goal in developing the interface should be to design the best problem-solving environment that non-intelligent technology will permit.

In the case of our programming application, this structured interface becomes the structured editor. We have developed a set of facilities for taking a BNF specification of the grammar of the language plus some prettyprinting information and automatically compiling the structured editor for that language. Thus, in the programming languages domain it is relatively painless to get a tutor for a new programming language provided one buys into this sort of interface.

In the case of the word problem tutor we want to create an interface that replicates the existing interface illustrated in Figure 4. This system consists of a set of facilities for bring up diagrams, labeling them, writing equations, and solving them. In particular, we provide the student with the power of a symbolic calculator and relieve them of the need to actually solve expression. Given that the interface already exists one might question the need to recreate it. Besides the fact that it only resides on a dying AI machine (the Xerox d-machine), it is necessary that the interface have hooks into the production system.

Insert Figure 4 About Here

Step 2: Specify Solution Syntax

It is necessary that the production system model be able to read the existing state of the solution and modify that solution state just as a student can. In a production system, access to all knowledge is through a working memory. This requires that we specify a syntax for representing the current screen structure in working memory. Each time an change takes place to the screen it is necessary to update the working memory's representation. In the case of programming, this is a hierarchical representation of the code structure. In the case of the word problem tutor it is a representation of the state of the diagram and of the equations with variables providing cross-references between corresponding quantities.

Step 3: Specify Syntax of Problem Representation

One must also keep a representation of the problem. In the case of certain formal domains like geometry or algebra symbol manipulation, the domain already has a set of formal conventions and notations for encoding problem states and one can represent the problem in those terms. Domains like programming or algebra word problems are much more difficult because they traditionally use informal English specification of the problem. An entirely "honest" problem representation would be as a string of words but this would involve us in the very difficult process of natural language understanding. It is also the case that the students really do not have difficulty in the language understanding components. So it is somewhat irrelevant to try to model the natural language processing. This motivates one to search for some internal representation that represents the immediate product of the natural language understanding process. In the case of word problems we hope to use something like the propositional representation that Kintsch and colleagues (Kintsch & Greeno, 1985) have used with some success to model word problem solution.

We have problems with such a propositional representation in the case of programming because there is the rather unbounded process of design in going from this to an algorithmic representation. Thus, we would like to have a representation that encodes the algorithm in some language-free way and then focus on tutoring the realization of that algorithm in some target programming language. This leaves open the goal of teaching algorithm design in certain restricted applications such as sorting algorithms.

We have chosen to represent the programming algorithms in an internal representation that is rather closely tied to prolog. There are a number of motivations for this choice. First, we have found, as have others (Taylor 1987), that as long as students' prolog programming involves treating the language declaratively and avoids dealing with procedural details like the cut it is a very easy language to use. Thus, it provides a somewhat natural algorithmic representation. It is also relatively easy to generate English descriptions again if we ignore procedural issues. However, this being said, the prolog choice is somewhat arbitrary but does provide us with a functional algorithmic representation. It does allow us the interesting potential of presenting the system with a problem in any language, have that problem compiled down to the internal representation, and then being able to tutor that problem in any target language.

Step 4: Production Rule Writing

The heart of the task is the production rule writing. With the interface specified and internally modelled and with the problem syntax specified, this becomes a relatively constrained task. This is to write production rules that map the current state of the problem solution and the problem representation into the interface actions that the ideal student should take. The constraints are helpful because they make the production rules fairly faithful to the task before the student. The idea that the production rules should produce interface actions is a deviation from many of the production system models that we have worked on in the past where the production rules actually specified changes to the problem structure. This is both unfaithful to what the student does and makes the production rules unnecessarily baroque in that it is often complex to specify changes to the solution structure. Now one can simply specify a interface action and have the consequence of that interface action in terms of solution representation automatically dumped into working memory.

One of the other features of our production system is that we are going to allow for unlimited number of productions to fire before an interface action is taken. This allows in many cases easier modelling but produces the prospect for serious temporary non-determinism as there may be many paths of these invisible productions taking place before an production fires that produces a disambiguating external action. The current production system is sufficiently efficient that we can deal with the bounded cases of such non-determinism that we anticipate in the programming and word problem applications. Of course, people could create systems in which there was an exponential explosion of non-determinism. However, our attitude to to leave this to developer's good sense rather than legislating the restrictions on invisible productions and resulting non-determinism.

Step 5: Attaching Declarative Information to Productions

Once the production rules have been written, one then needs to attach to these production rules examples for instruction and a set of questions that go with these examples. To date all of the instruction that we have associated with productions has been hand generated. We intend to explore an option of automatic generation of questions from the production form. However, we doubt that we will ever want to eliminate the option for handcrafting of instruction. This is one of the points of contact teachers like with the system--to have the ability to determine what is said.

Step 6: Entering and Annotating Problems

Another thing that teachers like is the facility to determine the problems that students see. The ideal mode for entry of such problems varies from domain to domain. In the programming domain, what is ideal is to be able to type in programs and have the tutor be able to tutor that program and equivalent solutions. As we said earlier this is something we have already developed. In addition to the program the teacher must enter a English problem statement plus attach English to various components of the code such as stating that a particular variable represents federal income tax.

In a domain like geometry it is easier to enter the formal problem statement and have the tutor solve it. The difference is that in programming it is the solution that is entered and in geometry it is the problem. The reason for this is that in programming it is easy to formalize solutions and for geometry it is easier to formalize the problem statement.

The word problem situation is similar to the geometry case. The teacher needs to enter a specification of the key mathematical relationships in the problem. However, in addition to this formal specification the teacher needs to be able to enter a natural language problem statement.

Step 7 Specifying the Curriculum

There are two ways in which the curriculum can be specified. One is just to identify the sequence of production rules to be taught. They need to be aggregated into small units that correspond basically to sections in typical textbooks. In principle this is all the tutor needs to compose a curriculum. It can choose from its stock of problems those that exercise the needed productions. However, teachers often have a desire to control the actual problems. So one might want to insist that certain prototypical problems be presented and specify the sequence in which these problems are presented. The tutor can then recruit additional remedial problems as needed. In may also be desired to enable the student to go on after a fixed set of problems even if they have not achieved mastery. This does not strike us as a wise educational policy but we even more firmly believe in the need to have the end user determine the shape of the system.

If there is going to be remediation and mastery-based learning, it is necessary to parameterize the system with quantities that reflect the ability level of the student and the difficulty of individual productions. These can be set at default levels to be adjusted automatically with experience with the student. Alternatively, the teacher may want to actively adjust the parameters that determine the remediation policy and when the student is judged to have mastered the material

Another adjustable dimension needs to be the coerciveness of instruction. As mentioned earlier our tutors are designed to move seemlessly from a tutor to a structured interface for problem solution and back again. This allows one to have a system which forces the student to stay on a solution path, which allows the student to do anything, or which provides the guidance we would recommend of the flag tutor. We intend to do further research on various possible configurations of coercion and freedom to provide an array of options for student and teacher.

What one gets for Free

We have described all the things that are necessary to create a tutor from scratch in this tutor. Therefore, it's important to note what comes with the system. One gets the production rule interpreter and facilities for production rule development. One gets the model tracer and with it facilities for presenting production-based instruction. One gets the mechanisms for knowledge tracing and curriculum sequencing. Finally, there are the facilities to fashion one's feedback mode and level instructional coercion. This is easily half the work in developing a tutor. In addition, if one is not starting from scratch, one can use existing work on the earlier steps in this sequence.

Summary

As advertised in the beginning this report has provided a mixture of past, present, and future. We have reviewed our past research that has led us to a particular view of how learning takes place and how tutoring can facilitate it. The key conclusions are that the productions define the critical units of learning and that the critical learning variable is to be number of problems solved and understood. A tutor should strive to maximize rate of problem solution and degree of understanding. We have in our tutoring work identified a particular minimalist tutoring style, called flag tutoring which achieves this.

We are currently doing explorations on generalizing the tutoring architecture in a way that reflects the outcome of our research experience. We described the multiple-languages tutor which is an approximation to the style of tutoring we would like to achieve. While it would be a gross exaggeration to say that our experiences with this tutor have been uniformly positive, it presents us with an number of achievements that make us confident for the future developments.

Our future plans for an authoring system reflect our desire to get the methodology into the hands of teachers and researchers. The major motivation for the emphasis on flexibility is to facilitate experimentation with the methodology. Enough pieces of the proposed authoring system are in place that we have begun reimplementing the multiple languages tutor. We hope to see the word problem tutor reimplemented within a year and in that time have a fairly general authoring system that can be distributed.

References

Anderson, J. R. (1983). *The Architecture of Cognition.* Cambridge, MA: Harvard University Press.

Anderson, J. R. (1987). Production systems, learning, and tutoring. In D. Klahr, P. Langley, & R. Neches (Eds.) , *Production system models of learning and development,* pp. 437-458. Cambridge, MA: MIT Press.

Anderson, J. R. (1990). *Cognitive Psychology and Its Implications. Third Edition.* New York: W. H. Freeman.

Anderson, J. R., Boyle, C. F., & Yost, G. (1985). The Geometry Tutor. In *Proceedings of IJCAI-85.* Los Angeles, CA: IJCAI, 1-7.

Anderson, J. R., Boyle, C. F., Corbett, A. T., & Lewis, M. W. (1990). Cognitive Modelling and Intelligent Tutoring. *Artificial Intelligence, 42,* 7-49.

Atkinson, R. C. (1972). Optimizing the learning of second-language vocabulary. *Journal of Experimental Psychology, 96,* 124-129.

Black, J. B., Bechtold, S., Mitrani, M., & Carroll, J. M. (1989). On line tutorials: What kind of inference leads to the most effective learning. In *Proceedings of the CHI '89 Conference on Human Factors in Computing Systems.* Boston, MA: Association for Computing Machinery.

Block, J. H. (1971). *Mastery Learning.* New York: Holt, Rinehart, & Winston .

Bovair, S., Kieras, D. E., & Polson, P. G. (1990). The Acquisition and Performance of Text-Editing Skill: A Cognitive Complexity Analysis. *Human Computer Interaction, 5,* 1-48.

Carroll, J. M. (In Press). *The Numberg funnel: Designing minimalist instruction for practical computer skill.* Cambridge, MA: MIT Press.

Carroll, J. M., & Mack, R. L. (1985). Metaphor, computing systems, and active learning. *International Journal of Man-Machine Studies,* pp. 39-57.

Corbett, A. T., & Anderson, J. R., (1990). The effect of feedback control on learning to program with the Lisp Tutor. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society* (pp. 796-806). Hillsdale, NJ: Erlbaum.

Corbett, A. T., Anderson, J. R., & Patterson, E. G. (In Press). Student modelling and tutoring flexibility in the LISP Intelligent Tutoring System. In C. Frasson and G. Gauthier (Eds.), *Intelligent tutoring systems: At the crossroads of artificial intelligence and education.* Norwood, NJ: Ablex.

Gray, W. D., & Atwood, M. E. (In Press). Transfer, adaptation, and use of intelligent tutoring technology: The Case of Grace. In M. Farr & J. Psotka (Eds.), *Intelligent Computer Tutors: Real World Applications.* New York, NY: Taylor & Francis.

Just, M. A., & Carpenter, P. A. (1987). *The Psychology of Reading and Language Comprehension.* Boston, MA: Allyn and Bacon, Inc.

Keedy, M. L, Bittinger, M. L., & Smith, S. A. (1978). *Algebra One.*. Menlo Park, CA: Addison-Wesley.

Kieras, D. E. (1982). A model of reader strategy for abstracting main ideas form simple technical prose. *Text, 2,* 47-82.

Kieras, D. E., & Bovair, S. (1986). The acquisition of procedures from text: A production-system analysis of transfer of training. *Journal of Memory and Language, 25,* 507-524.

Kintsch, W., & Greeno, J. G. (1985). Understanding and solving word arithmetic problems. *Psychological Review, 92,* 109-129.

Koedinger, K. R., & Anderson, J. R. (In Press). The role of abstract planning in geometry expertise. *Cognitive Science.*

McCloskey, M. (1983). Intuitive physics. *Scientific American, 248,* 122-130.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice Hall.

Nilsson, N. J. (1971). *Problem-solving methods in artificial intelligence.* New York:McGraw-Hill.

Reiser, B. J., Ranney, M., Lovett, M. C., & Kimberg, D. Y. (1989). Facilitating students reasoning with causal explanations and visual representations. In *Proceedings of the Fourth International Conference on Artificial Intelligence and Education.* Amsterdam.

Singley, M. K., & Anderson, J. R. (1989). *The transfer of cognitive skill.* Cambridge, MA: Harvard Press.

Singley, M. K., Anderson, J. R., & Gevins, J. S. (March 1990). Promoting Abstract Strategies in Algebra Word Problem Solving. (RC15861 (#69354)).

Sleeman, D., & Brown, J. S. (Eds). (1982). *Intelligent tutoring systems.* New York: Academic Press.

Sleeman, D., Kelly, A. E., Marlinak, R., Ward, R. D., & Moore, J. L. (1989). Studies of diagnosis and remediation with high-school algebra students. *Cognitive Science, 13,* 551-568.

Taylor, J. (1987). *Programming in Prolog: An in-depth study of problems for beginners learning to program in Prolog.* Doctoral dissertation, University of Sussex. Cognitive Studies Programme.

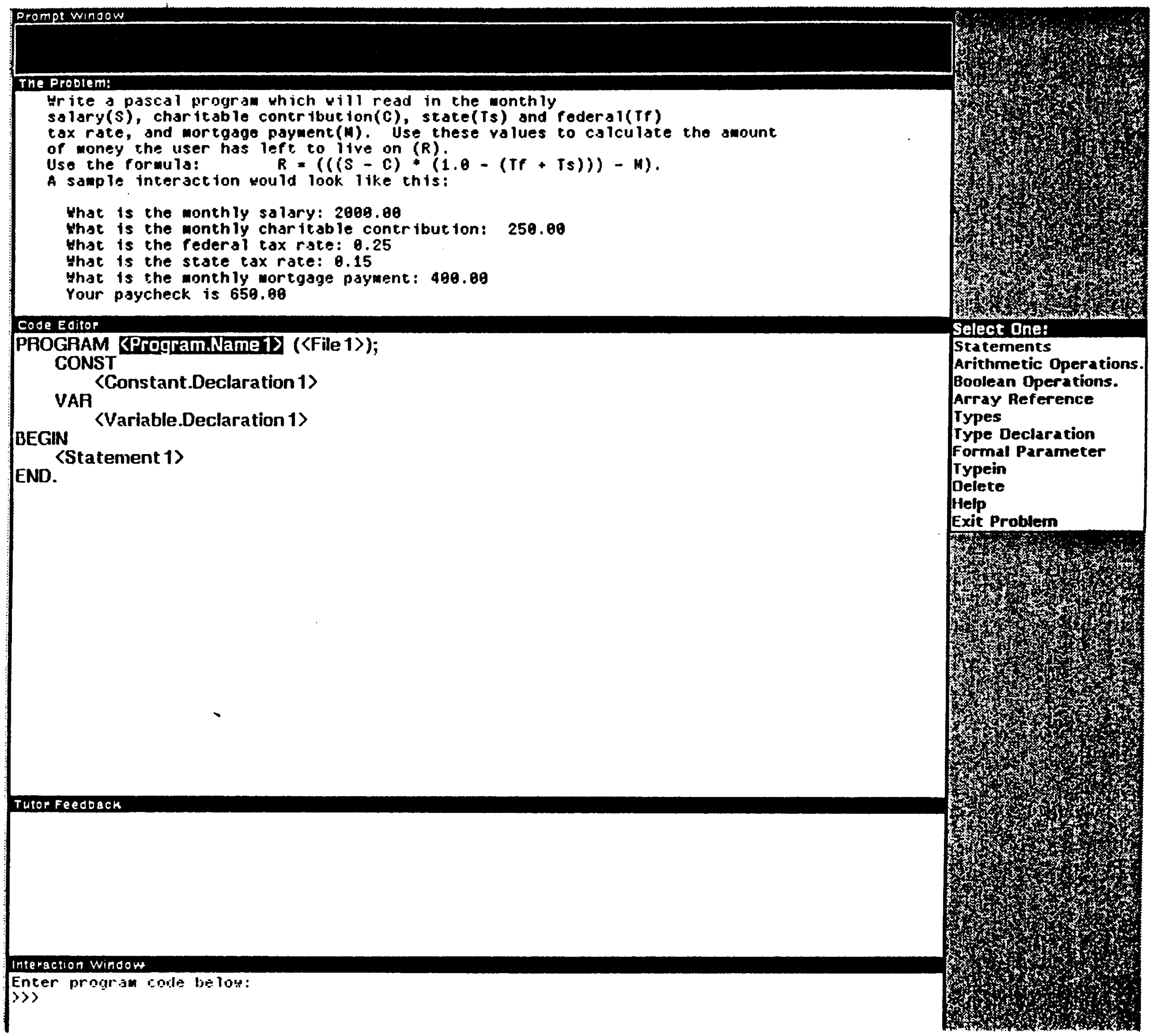
Figure Captions

Figure 1 The screen at the beginning of a PASCAL problem in the multiple-programming-languages tutor.

Figure 2 The screen in the midst of a PASCAL problem in the multiple-programming-languages tutor.

Figure 3 The progress window in the multiple-programming-languages tutor show level of mastery of the individual productions being monitored in the LISP lesson on recursion.

Figure 4 The screen image for the word-problem tutor showing problem statement (top), equations window (left), operations and variable window (middle), and diagram (right).

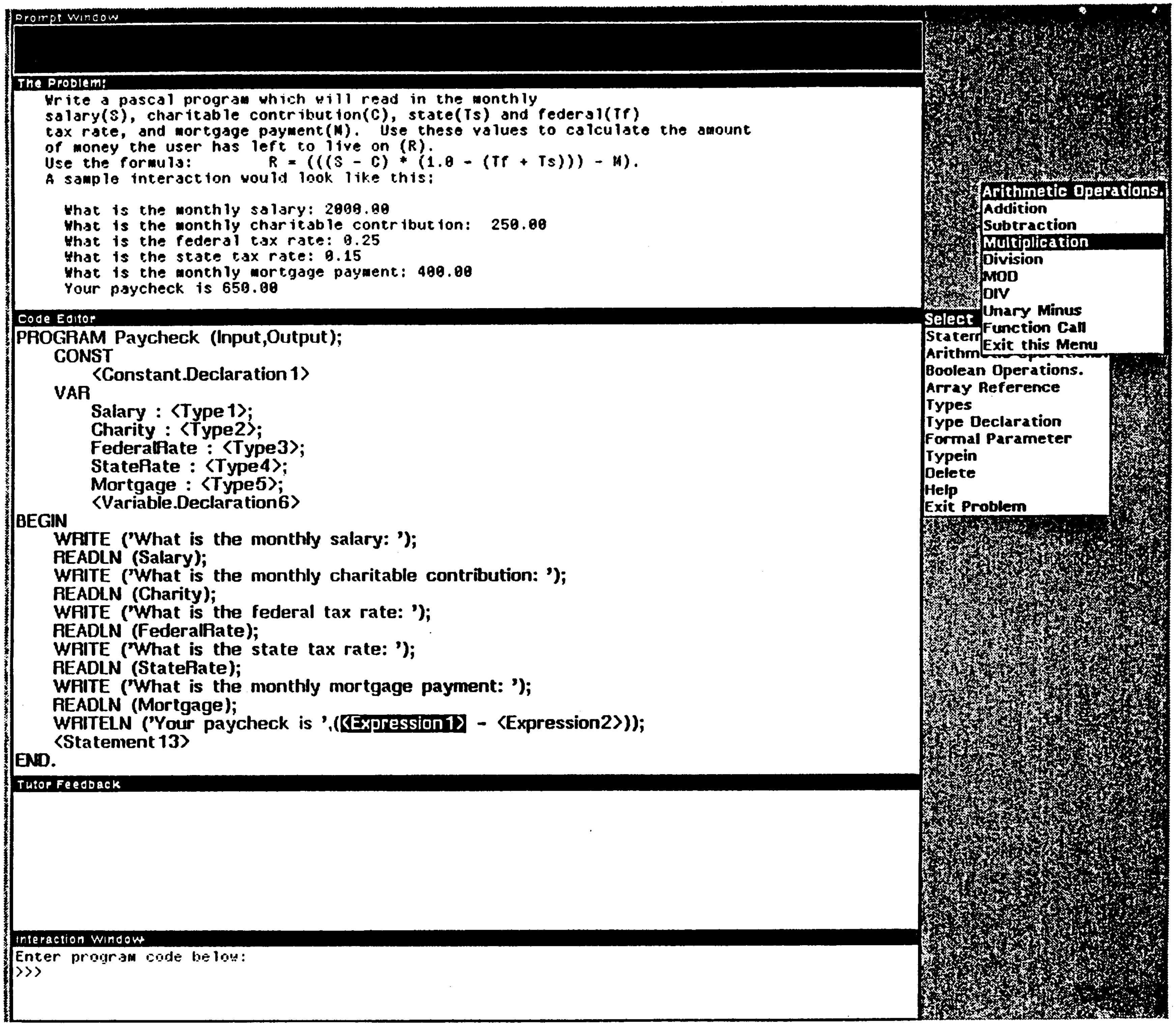


```
    Write a pascal program which will read in the monthly
    salary(S), charitable contribution(C), state(Ts) and federal(Tf)
    tax rate, and mortgage payment(M).  Use these values to calculate the amount
    of money the user has left to live on (R).
    Use the formula:        R = (((S - C) * (1.0 - (Tf + Ts))) - M).
    A sample interaction would look like this:

      What is the monthly salary: 2000.00
      What is the monthly charitable contribution:  250.00
      What is the federal tax rate: 0.25
      What is the state tax rate: 0.15
      What is the monthly mortgage payment: 400.00
      Your paycheck is 650.00
```

```
PROGRAM <Program.Name1> (<File1>);
    CONST
        <Constant.Declaration1>
    VAR
        <Variable.Declaration1>
BEGIN
    <Statement1>
END.
```

```
Enter program code below:
>>>
```

Figure 1



The Problem:

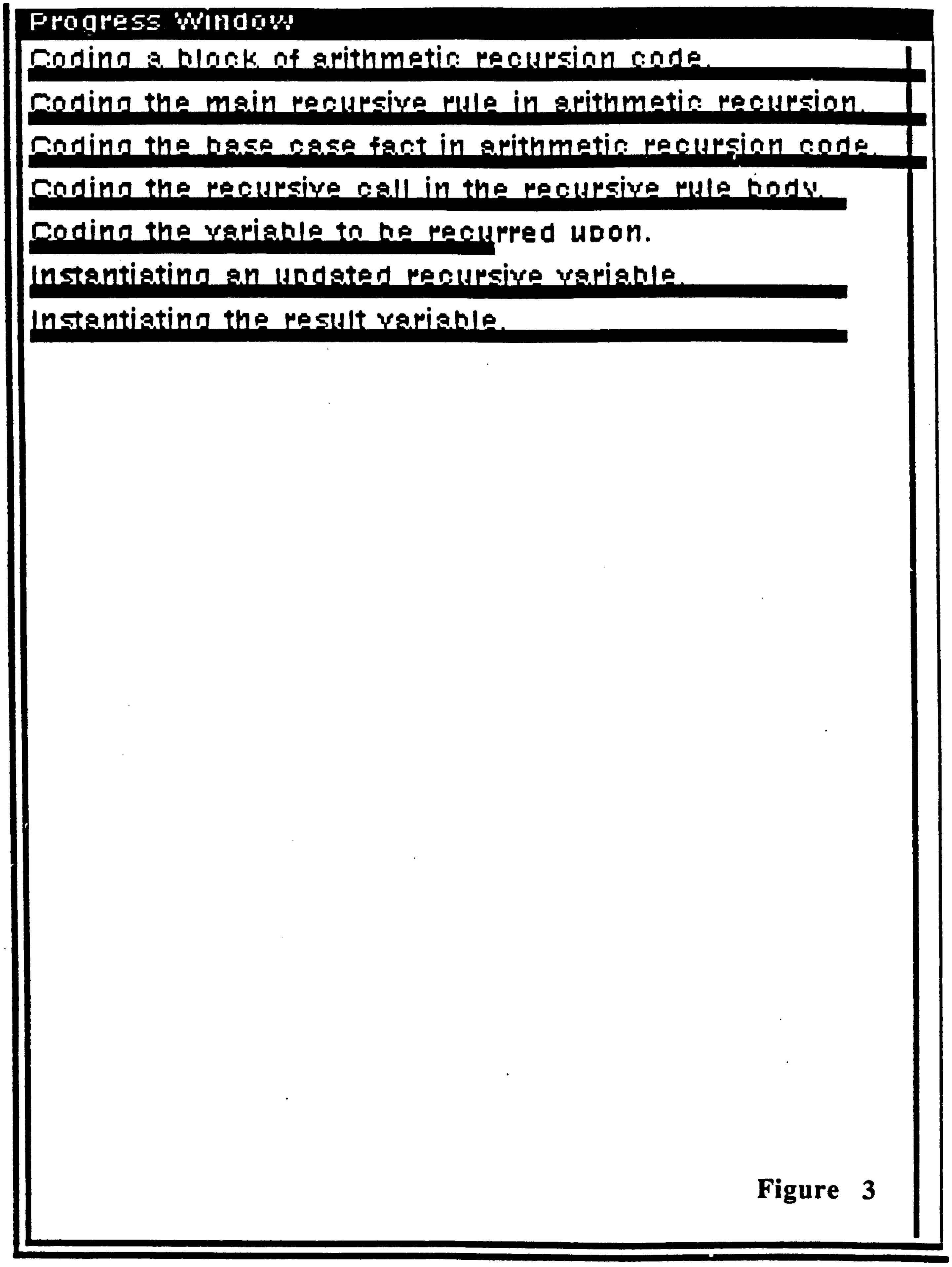
```
Write a pascal program which will read in the monthly
salary(S), charitable contribution(C), state(Ts) and federal(Tf)
tax rate, and mortgage payment(M).  Use these values to calculate the amount
of money the user has left to live on (R).
Use the formula:        R = (((S - C) * (1.0 - (Tf + Ts))) - M).
A sample interaction would look like this:

  What is the monthly salary: 2000.00
  What is the monthly charitable contribution:  250.00
  What is the federal tax rate: 0.25
  What is the state tax rate: 0.15
  What is the monthly mortgage payment: 400.00
  Your paycheck is 650.00
```

Code Editor

```
PROGRAM Paycheck (Input,Output);
    CONST
        <Constant.Declaration1>
    VAR
        Salary : <Type1>;
        Charity : <Type2>;
        FederalRate : <Type3>;
        StateRate : <Type4>;
        Mortgage : <Type5>;
        <Variable.Declaration6>
BEGIN
    WRITE ('What is the monthly salary: ');
    READLN (Salary);
    WRITE ('What is the monthly charitable contribution: ');
    READLN (Charity);
    WRITE ('What is the federal tax rate: ');
    READLN (FederalRate);
    WRITE ('What is the state tax rate: ');
    READLN (StateRate);
    WRITE ('What is the monthly mortgage payment: ');
    READLN (Mortgage);
    WRITELN ('Your paycheck is ',(<Expression1> - <Expression2>));
    <Statement13>
END.
```

Interaction Window

```
Enter program code below:
>>>
```



Progress Window

Coding a block of arithmetic recursion code.

Coding the main recursive rule in arithmetic recursion.

Coding the base case fact in arithmetic recursion code.

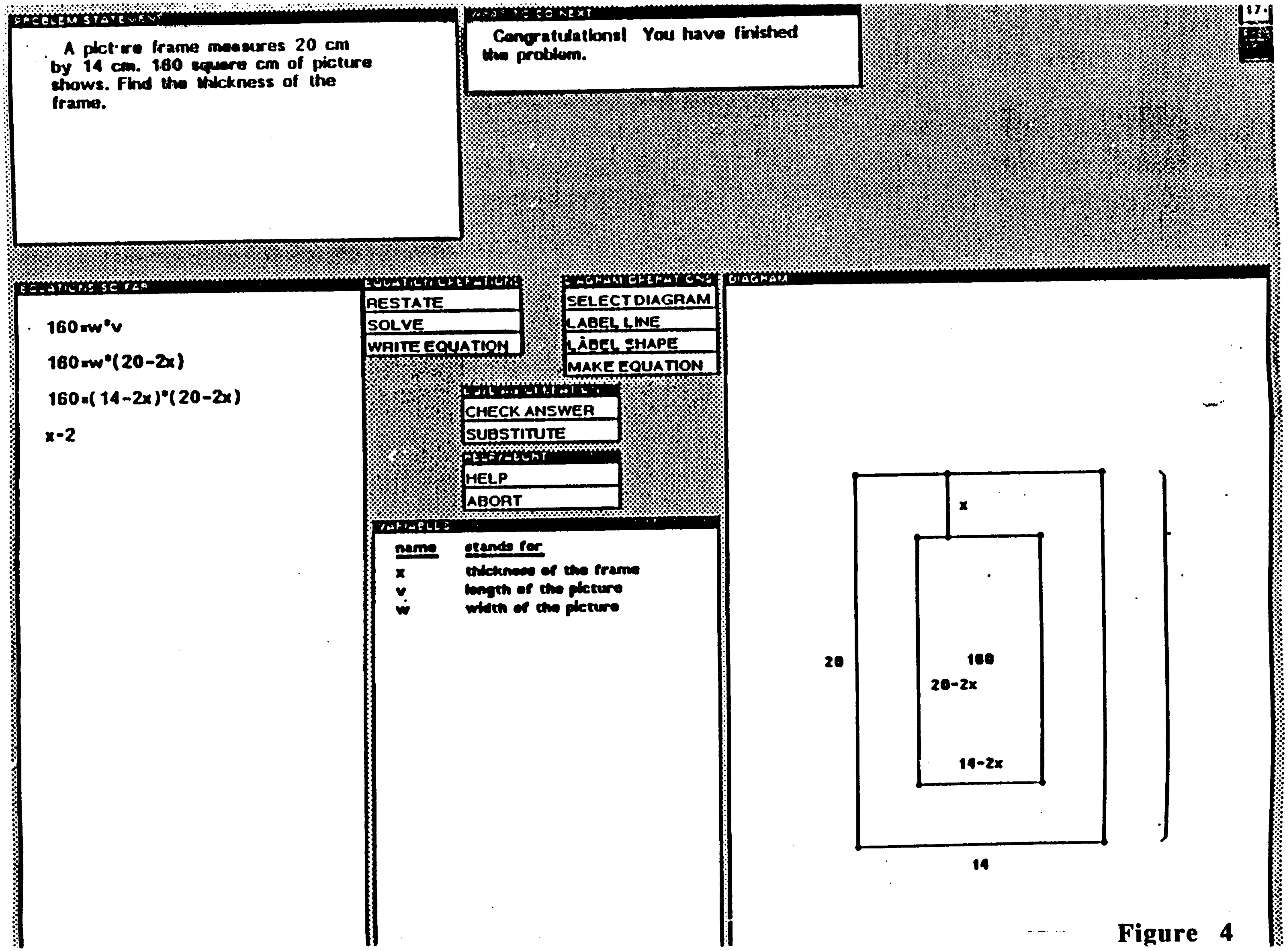
Coding the recursive call in the recursive rule body.

Coding the variable to be recurred upon.

Instantiating an updated recursive variable.

Instantiating the result variable.

Figure 3



A pict·re frame measures 20 cm by 14 cm. 160 square cm of picture shows. Find the thickness of the frame.

Congratulations! You have finished the problem.

160=w*v

160=w*(20-2x)

160=(14-2x)*(20-2x)

x-2

name	stands for
x	thickness of the frame
v	length of the picture
w	width of the picture

Figure 4